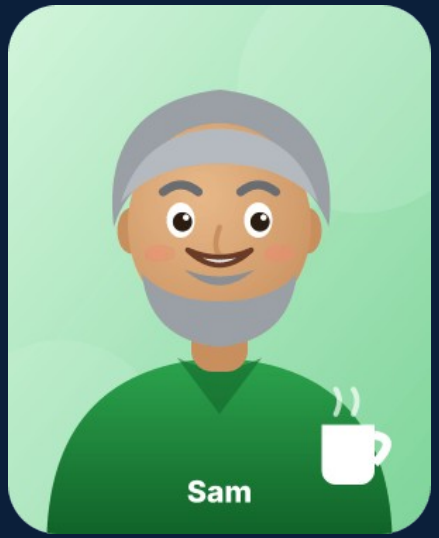
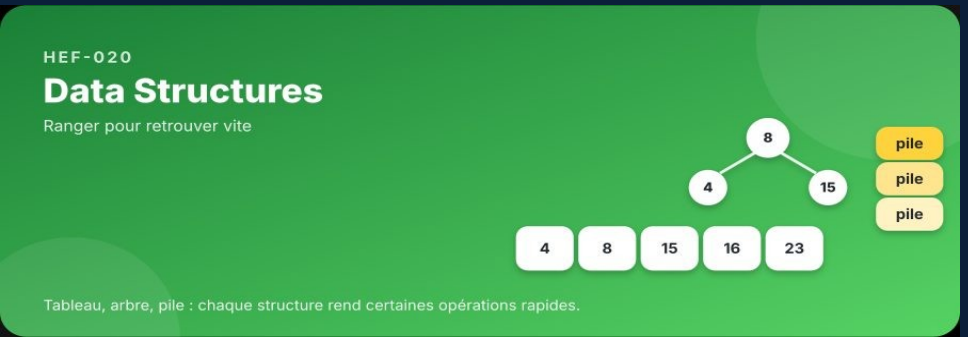
HEF-020 · MISSION 03 / 10

# Piles et files

*Dernier arrivé, premier sorti… ou premier arrivé, premier sorti. Deux règles, mille usages.*




**Badge : Maître de la file**

70 min · débutant

L'histoire

# L'histoire

L'éditeur de HashBox a besoin d'un bouton « **Annuler** ». Et le service d'envoi des e-mails doit traiter les messages dans l'ordre où ils arrivent.




> *« Ce sont deux structures différentes avec deux règles différentes, dit Sam. Une pile pour l'annulation, une file pour les e-mails. »*

**Dans cette mission, tu apprends deux structures simples qui sont partout dans les logiciels.**

Ta mission

# À la fin, je serai capable de…

1. décrire la règle d'une pile (LIFO) et d'une file (FIFO)
2. implémenter une pile et une file en Python
3. utiliser une pile pour vérifier des parenthèses
4. reconnaître les usages réels de chacune
5. expliquer pourquoi `list.pop(0)` est un mauvais choix pour une file

Défi d'échauffement

# Défi d'échauffement

Pense à une **pile d'assiettes** et à une **file d'attente à la boulangerie**. Dans chaque cas : où ajoute-t-on ? D'où retire-t-on ? Quelle personne ou assiette sort en premier ? Trouve un exemple informatique pour chacune.

Le concept

# La pile (stack) — LIFO

**LIFO** : *Last In, First Out*. Le dernier arrivé est le premier sorti.

Opérations : `empiler` (ajouter au sommet), `depiler` (retirer le sommet), `sommet` (regarder sans retirer), `est_vide`.

Usages : bouton « Annuler », historique de navigation, appels de fonctions (la pile d'appels de la récursion !), vérification de parenthèses.

```
class Pile:
    def __init__(self):
        self._elements = []
    def empiler(self, x):
        self._elements.append(x)
    def depiler(self):
```

Le concept

# La pile (stack) — LIFO (suite)

```
        if not self._elements:
            raise IndexError("pile vide")
        return self._elements.pop()
    def sommet(self):
        return self._elements[-1]
    def est_vide(self):
        return len(self._elements) == 0
```

## La file (queue) — FIFO

**FIFO** : *First In, First Out*. Le premier arrivé est le premier servi.

Opérations : `enfiler` (ajouter à l'arrière), `defiler` (retirer de l'avant), `est_vide`.

Usages : file d'impression, envoi d'e-mails, requêtes à traiter, parcours en largeur (on y reviendra).

Le concept

# La file (queue) — FIFO (suite)

En Python, on utilise `collections.deque` : ajouter et retirer aux deux extrémités sont rapides.

```
from collections import deque

class File:
    def __init__(self):
        self._elements = deque()
    def enfiler(self, x):
        self._elements.append(x)
    def defiler(self):
        if not self._elements:
            raise IndexError("file vide")
        return self._elements.popleft()
    def est_vide(self):
        return len(self._elements) == 0
```

Le concept

# La file (queue) — FIFO (suite)

*⚠ liste.pop(0) retire au début d'une liste Python mais décale tous les éléments : lent sur de grandes files. Préfère deque.*

## Parenthèses équilibrées

Pour vérifier `([]{})` : on empile les ouvrantes, et à chaque fermante on dépile en vérifiant qu'elles correspondent. À la fin, la pile doit être vide.

Exemple pas à pas

# Exemple pas à pas

**Problème** : le texte `"{[()]}"` est-il bien parenthésé ? Et `"{[(])}"` ?

```
def equilibre(texte):
    paires = {")": "(", "]": "[", "}": "{"}
    pile = []
    for c in texte:
        if c in "([{":
            pile.append(c)
        elif c in ")]}":
            if not pile or pile.pop() != paires[c]:
                return False
    return len(pile) == 0

print(equilibre("{[()]}"))   # True
print(equilibre("{[(])}"))   # False
```

**Trace pour `{[(])}` :**

Exemple pas à pas

# Exemple pas à pas (suite)

| Lu | Action | Pile |
|---|---|---|
| { | empiler | { |
| [ | empiler | { [ |
| ( | empiler | { [ ( |
| ] | dépiler ( : ≠ [ | échec ! |

**Pourquoi la pile ?** La dernière ouvrante doit être la première fermée : c'est exactement LIFO.

# À toi, avec aide

Implémente un **historique avec Annuler/Rétablir** : `Editeur` avec `ecrire(texte)` (ajoute au contenu), `annuler()` (retire la dernière écriture), `contenu()`.

Utilise une pile des écritures. Teste : écrire "a", "b", "c" → contenu "abc" ; annuler → "ab".

À toi, avec aide · réponse

# Indices progressifs

## Indice 1

Garde une pile des textes écrits.

## Indice 2

`annuler` dépile le dernier texte ; si la pile est vide, ne fait rien.

## Indice 3

`contenu` concatène les éléments de la pile.

À toi, avec aide · réponse

# Réponse

```
class Editeur:
    def __init__(self):
        self._ecritures = []

    def ecrire(self, texte):
        self._ecritures.append(texte)

    def annuler(self):
        if self._ecritures:
            self._ecritures.pop()

    def contenu(self):
        return "".join(self._ecritures)

e = Editeur()
for t in ("a", "b", "c"):
    e.ecrire(t)
e.annuler()
print(e.contenu())   # ab
```

# À toi, en solo

Simule le **guichet de support** de HashBox avec une file : `Guichet` avec `arriver(nom)`, `servir()` (renvoie le nom du suivant ou `None`), `en_attente()` (nombre).

Puis écris `simuler(arrivees, services)` où `arrivees` est une liste de noms et `services` un nombre de personnes servies, et renvoie la liste des servis dans l'ordre.

À toi, en solo · réponse

# Pistes de correction

```
from collections import deque

class Guichet:
    def __init__(self):
        self._file = deque()
    def arriver(self, nom):
        self._file.append(nom)
    def servir(self):
        return self._file.popleft() if self._file else None
    def en_attente(self):
        return len(self._file)

def simuler(arrivees, services):
    g = Guichet()
    for nom in arrivees:
        g.arriver(nom)
    servis = []
    for _ in range(services):
        s = g.servir()
```

# Pistes de correction (suite)

```
        if s is not None:
            servis.append(s)
    return servis

print(simuler(["Ada", "Bob", "Cléo"], 2))   # ['Ada', 'Bob']
```

# Pièges classiques

| Piège | Pourquoi c'est un problème | Comment l'éviter |
|---|---|---|
| **Dépiler une pile vide** | `IndexError` | Tester `est_vide` ou lever une erreur claire |
| **Utiliser `pop(0)` pour une grande file** | Très lent | Utiliser `deque` |
| **Confondre LIFO et FIFO** | Ordre de traitement inversé | Dessiner un exemple avant de choisir |
| **Oublier de vérifier la pile vide à la fin** | `"("` passerait pour correct | Toujours `len(pile) == 0` à la fin |

Mini-boss

# Mini-boss

Écris un **évaluateur d'expressions en notation polonaise inverse (RPN)** : `"3 4 + 2 *"` se lit « 3 plus 4, fois 2 » = 14.

Règle : on lit les éléments ; un nombre est empilé ; un opérateur (`+ - * /`) dépile deux nombres, calcule et empile le résultat.

1. Écris `evaluer_rpn(texte)`.
2. Teste `"3 4 + 2 *"` → 14, `"5 1 2 + 4 * + 3 -"` → 14.
3. Gère les erreurs : pile insuffisante, division par zéro (message clair).

Mini-boss · réponse

# Pistes de correction

```
def evaluer_rpn(texte):
    pile = []
    for jeton in texte.split():
        if jeton in "+-*/" and len(jeton) == 1:
            if len(pile) < 2:
                raise ValueError("expression invalide")
            b, a = pile.pop(), pile.pop()
            if jeton == "+": pile.append(a + b)
            elif jeton == "-": pile.append(a - b)
            elif jeton == "*": pile.append(a * b)
            else:
                if b == 0:
                    raise ZeroDivisionError("division par zéro")
                pile.append(a / b)
        else:
            pile.append(float(jeton))
    if len(pile) != 1:
        raise ValueError("expression invalide")
    return pile[0]
```

Mini-boss · réponse

# Pistes de correction (suite)

```
print(evaluer_rpn("3 4 + 2 *"))            # 14.0
print(evaluer_rpn("5 1 2 + 4 * + 3 -"))    # 14.0
```

Attention à l'ordre `b, a = pile.pop(), pile.pop()` : le premier dépilé est l'opérande de droite.

Quiz éclair

# Quiz éclair

**1.** Une pile suit la règle :

A. premier arrivé, premier sorti
B. dernier arrivé, premier sorti
C. aléatoire

**2.** Une file suit la règle :

A. dernier arrivé, premier sorti
B. aléatoire
C. premier arrivé, premier sorti

**3.** Pour vérifier des parenthèses, on utilise :

A. une pile

Quiz éclair

# Quiz éclair (suite)

B. une file
C. un tri

**4.** 🔁 *Révision mission 02* : Insérer au début d'un grand tableau est :

A. gratuit
B. coûteux (décalage)
C. impossible

**5.** 🔁 *Révision mission 01* : Un ADT se décrit par :

A. son code interne
B. sa couleur
C. ses opérations

Quiz éclair · réponse

# Réponses

1. **B**.
2. **C**.
3. **A**.
4. **B**.
5. **C**.

# Je réfléchis

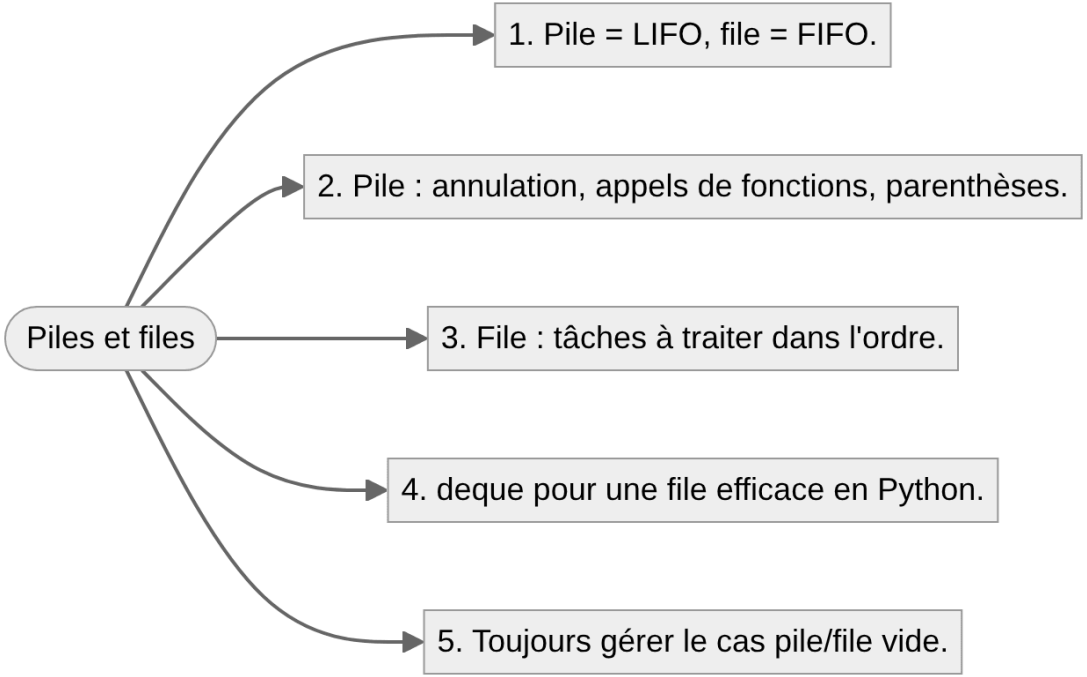
1. Où, dans ta journée, vis-tu une file ? Une pile ?
2. Quelle fonctionnalité de tes applications préférées repose sur une pile ?

# À retenir

1. Pile = LIFO, file = FIFO.
2. Pile : annulation, appels de fonctions, parenthèses.
3. File : tâches à traiter dans l'ordre.
4. deque pour une file efficace en Python.
5. Toujours gérer le cas pile/file vide.

À retenir

# Le schéma de la mission



MISSION ACCOMPLIE

# +100 XP

## Badge débloqué : Maître de la file

Prochaine mission : Tables de hachage : retrouver sans chercher

